Journal of Physics: Conference Series **331** (2011) 052028 doi:10.1088/1742-6596/331/5/052028

Deska: Tool for Central Administration of a Grid Site

Jan Kundrát, Martina Krejčová, Tomáš Hubík, Lukáš Kerpl

E-mail: kundratj@fzu.cz
Institute of Physics, AS CR, Na Slovance 22, 182 21, Prague, Czech Republic

Abstract. Running a typical Tier-2 site requires mastering quite a few tools for fabric management. Keeping an inventory of installed HW machines, their roles and detailed information, from IP addresses to rack locations, is typically done using various in-house applications ranging from simple spreadsheets to web applications. Such solutions, whose documentation usually leaves much to be desired, typically do not prevent a significant duplication of information, and therefore the data therein quickly become obsolete.

After having deployed Cfengine as one of a few sites in the WLCG environment, the Prague Tier-2 site set forth to further automate the fabric management, developing the Deska project. The aim of the system is to provide a central place to perform changes, from adding new machines or moving them between racks to changing their assigned service roles and additional metadata. The database provides an authoritative source of information from which all other systems and services (like DHCP servers, Ethernet switches or the Cfengine system) pull their data, using newly developed configuration adaptors. An easy-to-use command line interface modelled after the Cisco IOS-based switches was developed, enabling the data center administrators to easily change any information in an intuitive way.

We provide an overview of the current status of the implementation and describe our design choices aimed at further reducing the system engineers' workload.

1. Background

One of the activities taking place at the Institute of Physics of the AS CR (FZU) [1] is running a local Tier-2 computing center connected to an international WLCG computing grid, supporting user programs from various scientific communities from the whole world including the D-0 experiment in Fermilab and the famous LHC accelerator from CERN. In December 2010, the data center consisted of roughly 350 machines, with more hardware to arrive in early 2011. All these resources were managed by just a handful of system administrators. Additional hardware procurements were in the process and the overal installed capacity had been steadily growing over the years, but the capacity of the administration team stagnated at roughly three full-time equivalents, with no matching increase in the number of staffers planned in the intermediate future [2].

Given the rather high machines-per-administrator ratio, the staffers were exploring how to make the system administrators' life easier and less error prone. Duplication of information was identified as a problem significantly contributing to the stress levels and occasional operational issues.

2. Duplication of Information

Consider the typical scenario when new machines arrive to the data center and have to be installed and put into production. The information about each node has to be scattered into smaller chunks and stored in the configuration of core networking services; the machines have to be configured and added into various monitoring appliances. In the end, a record for each machine has to be kept in at least the following places:

- HW inventory DB
- Warranty & issue tracking
- Switch port configuration
- DHCP server
- DNS
- Cfengine [3] roles
- Torque's CPU multipliers
- MRTG [4] & RRD network graphs
- Nagios [5]
- Ganglia [6]
- Munin [7]
- ...

Some of these places are notoriously prone to omission from the update process, leading to eventual inconsistencies. It can also be frustrating for the staffers to duplicate work over and over again by iterating over a (usually under-documented) checklist. A central place storing information about all machines in the network is clearly missing and none of the existing tools marketed as inventory managers were deemed suitable for this task. Therefore, it has been decided that the Institute shall fund a development of a tool which will *automate the configuration process* and focus on bringing the control back to a central place.

3. Analysis of Existing Tools

The Institute has been using an internal web application for general hardware inventory management, but without any integration with the rest of the management stack. The database contained information about IP addresses, hostnames and network topology, but these pieces of information weren't actually utilized throughout the network. The staffers were reportedly having issues with the user interface provided by the web application. Being accustomed to the Unix way of doing things over a command line interface, using a web application felt like a step backwards.

Most of the existing third-party tools, proprietary or free software, again focus on the web side of affairs; it is rare to find an appliance which offers a scripting API. Some popular choices, like the OCS Inventory [8], seem to focus on auto-discovery of heterogenous equipment, a hard task which is however not important for the Institue. Others, like the not-so-known Smurf [9], do offer an API to use in scripting, but are not actively marketed as "ready for use by third parties" or undergo an internal rewrite in order to elliminate a dependency on a proprietary RDBMS implementation. It's also rather surprising to routinely come across tools like the Rackmonkey [10] which cannot deal with machines with different form factors than traditional "pizza boxes", such as blade servers or SGI twins. Furthermore, most of the inventory management applications are aimed mainly at automatic discovery of very heterogeneous environment, which is not the most serious issue the Institute is facing.

Finally, even the proprietary solutions, like the System Director from IBM, offer only a limited subset of desired functionality. While it's clear that certain features like automatic discovery

and tight integration with vendor's low-level fabric management tools are a clear benefit, these advantages are of little value when the basic functionality is simply lacking.

Considering all points mentioned above, we maintain that a new tool is indeed needed at this point, simply because no already existing system offers all key features we require. Therefore, the Deska suite shall focus on general usability, shall offer a CLI access, integrated revision control for all records stored in the database, support scripting and be generic enough to allow future extensions.

4. Architecture of Deska

The goal of the project is to develop a database storing generic description of all the resources that a Grid site is using; such a database should be capable of describing hardware infrastructure of a whole data center, from physical machines with corresponding part numbers and rack locations to the network interconnects, coupling of operating system instances (aka "hosts") to physical machines as well as logical relations between various services and their dependencies. This database then acts as the single authoritative source of information for generating configuration of all components in the system.

4.1. Generic Database

As the history has proven that it is extremely hard to give accurate predictions about future developments and trends in computing, the database does not impose arbitrary restrictions to its data model. The core DB code only assumes that it is working with *collections of objects* and some *relations between them*. Actually defining a usable *scheme* of the database is a separate problem from the general database design. The database supports linear *version control* of the records contained therein.

This component of Deska is based on a PostgreSQL server wrapped with an interface implementing a Deska-specific JSON-based API. Additional business logic for interactively providing a feedback of the impact of performed changes to the administrator is implemented on this level, too.

4.2. Management Interface

Being the first part of Deska which is actually visible to the regular Unix system administrator, the application interface is similar to a CLI-oriented interface used on enterprise-grade Ethernet switches, notably to the Cisco IOS shell. Such a format allows storing of plaintext dumps of the database in a version control system for auditing purposes.

The CLI interface works on the same level of abstraction as the generic database, that is, it does not contain any specific knowledge of the database scheme being used. All information required for the operation of the CLI is retrieved via the Deska's database API, and no changes to the CLI source code are needed when the database scheme changes.

In future, the CLI interface will include "wizards" for accelerating common tasks like deploying a new machine. A strong emphasis is given on usability, and the CLI will support a non-interactive mode of operation suitable for scripting and batched updates.

4.3. Database Scheme

The database scheme is a rather abstract part of the Deska DB, but an important one nonetheless. This scheme must be extensible to allow installing future pieces of equipment without large-scale code changes, shall try to anticipate future trends in hardware development, but also attain a reasonable simplicity allowing people to use it without significant pain. The goal here is not to achieve purity at any cost, but implement a solution which is easy to deploy, use and maintain.

4.4. Add-on Modules

The information contained in the database shall be actually put into real use. Towards that goal, a set of components generating configuration will be provided. These modules will consult the Deska API and will be used for templating configuration for all core network services currently deployed at the Institute, namely for Nagios, the BIND name service demon, DHCP server and Ganglia and Munin master servers. The design of the API reflects generic needs of any data center to allow eventual deployment of new components, should the need arise. All modules will use only the public and documented API for retrieving all the data they need.

These modules will be invoked when an administrator decides to push her changes to the central database. The Deska server will present her with a list of changes performed by herself along with the proposed new variant of the generated files. This way, she can assess the scope of changes she has just performed, greatly reducing the potential of a configuration error knocking down the whole infrastructure.

In addition, tools will be written to perform consistency checks between the state recorded in the database and the actual system shape where the automatic configuration generation is not feasible for safety reasons. A classic example of such a system are network switches where network administrators are rather leery of automated configuration because a single error can cause large network segments to separate.

5. Database Structure

The core Deska database deals exclusively with flat lists of objects, with each list sharing objects of a common type. An object in this case refers simply to an equivalent of the C language `struct`, that is, a data structure containing list of attributes. Each attribute has a name and an associated primitive data type like a string, an integer, a floating point number or an IP address. Defining additional "primitive types" is possible, but requires modifying the Deska's source code. This way, the database layer which is responsible for rather low-level tasks like maintaining proper revision control does not have to deal with complicated inter-relations between objects. The only methods implemented at the lowest layer are targeted at manipulating individual objects in the lists and performing queries against them. An overview of the implemented methods is available at `src/deska/db/Api.h` in the Deska distribution [11].

5.1. Abstract and Concrete Objects, Inheritance and Templates

Suppose a typical database scheme for a data center. It should certainly keep track of individual machines (the physical objects in the racks), but also abstract away certain features which are defined by the brand or model of each machine. For example, it would be a waste of time having to specify rack height of 1U for each server instead of simply stating that it's a HP DL-360. Therefore, the database has to offer a way of *referencing* an object of a different kind, and it makes sense to describe hardware models in a separate object pool than the physical machines. In the default database scheme shipped with Deska, the hardware models are of type `hwmodel` while the physical machines are of the type `host`.

Further down this road, it's obvious that some models share certain properties, like their form factor. It could therefore be interesting for the database to support *specialization* of entries, where objects of the same kind have the ability to inherit certain properties from a *template* and override values which have to differ. Again, the lower layer of the database abstraction does not have to deal with actually interpreting these values, it just has to offer a way to report this information to the upper layers. The Deska DB supports this specialization by the keyword `template`.

The following is an example of how the textual dump of the database could look like. We demonstrate all of the advanced concepts, like using the `template` keyword for specializing

a generic definition on line 7, referencing a foreign object kind on line 17 and embedding an additional object on lines 20 and 21.

```
1  hwmodel sgi-twin
2      in generic-rack
3      occupies height 1
4  end
5
6  hwmodel xe310
7      template sgi-twin
8      benchmark hepspec 68
9      cpu ``E5420''
10     ram 16GB
11     sockets 2
12     cores-per-socket 4
13     disk "SAS 300GB"
14 end
15
16 host salix01
17     hw xe310
18     serial X0008274
19
20     interface eth0 mac 00:30:48:C5:42:BE net wn-nat ip 172.16.1.1
21     interface bmc shared-port eth0 net monitoring ip 192.168.10.1
22
23     role wn
24 end
```

Please note that the above example is rather limited and not self-contained; it is intended merely as a demonstration of how the configuration looks like. A much more detailed example is available in the Deska source repository [11] in file `doc/cli/demonstration`.

5.2. Building Friendly Scheme

While the design described so far is without any doubt beneficial for the database implementation, it leaves much to be desired when human users come into play. Therefore, we have to build on these foundations and offer a rich, scalable layer allowing users to work with these records without undertaking extra pain. One example of such an extension can be well illustrated with network interfaces. For brevity, this article simplifies matters a bit and assumes that there is a one-to-one mapping between network addresses and network interfaces, which is not true, especially in a modern data center.

Clearly, any machine can trivially have multiple network interfaces, maybe one connected to a public network and one to a management one. Because the underlying database manages only flat lists of objects, one would have to explicitly create a brand new object for each interface of each machine, greatly inflating the effort needed when deploying new machines. That's when the upper layers of Deska come into play. While the dedicated object for an interface indeed gets created under the hood, this complexity is hidden by the UI and the interface is *embedded* into the host to which it logically belongs. This is a natural concept which corresponds to the real world we are modelling.

When querying the database scheme, the API shall therefore inform the upper layers that there is a logical link between the instances of the `host` type and an `interface` type, and the relation is of a special kind which allows embedding the interface definition to the host section.

Similar relations for one-to-one mapping, as commonly found between physical machines and the running OS image when not using virtualization, are also supported.

5.3. Defining and Extending the Scheme

The Deska's motto is a meme *don't repeat yourself* and it proclaims to reduce the information duplicity. It would be rather hypocrite to propose yet another domain-specific language for describing the real database scheme, as there already is a perfectly ubiquitous language for describing tabular layouts, the SQL. Deska is therefore shipped as a set of stored procedures and accompanying scripts which merely look at the pre-existing structure of an SQL database, using PostgreSQL's facilities for run-time scheme inspection, and pre-generate the database code for dealing with individual scheme being used on-site.

It is clear that the table layout cannot be arbitrary, simply because of the sheer complexity of such a design. We therefore have to impose a set of restrictions on the database scheme. We start with defining a primary key to always act as a "name" of an item. All other columns have to be of one of a pre-defined set of types, and their name will directly correspond to the name of the attribute they are modelling.

Database integrity constraints are fully supported; the only requirement is that they have an explicitly assigned name which encodes the type of the modelled relation. When the constraints have these names, Deska can automatically extract the relationship information and pass it further via its API.

6. Performance Considerations, Scalability and Reliability

The Deska architecture strongly prefers simple and effective solutions over complicated and over-engineered alternatives. For example, instead of patching a DHCP server to dynamically retrieve its configuration data from the Deska server via its JSON API, possibly upon each request, it is assumed that these configuration files are regenerated during each commit and then pushed to the relevant services, all via existing configuration management utilities. This approach makes the whole distributed system much more resilient against possible interruptions and service disruptions, and substantially reduces the amount of queries that actually hit the PostgreSQL database server. Hence, the overal load of the system is no longer a function of all nodes "consuming the data", but instead follows the frequency (and amount) of database modifications.

Certain operations, like retrieval of object history, still have potential to easily overload the SQL server due to the sole amount of data they have to process. Any such operation is, however, only available to authorized users, and can therefore be easily limited by proper configuration.

Any "hard limits" in terms of attributes-per-object or number of objects are virtually non-existent, as they depend on any applicable PostgreSQL table space constraints. The Deska DBAPI for object retrieval supports partial fetches, and is therefore not considered a bottleneck. The only limitation imposed at this level is that each response of the server is read into one contiguous block of memory by the C++ implementation of the Deska DBAPI. However, on a reasonable hardware, the Deska suite shall have no problems with hundreds of thousands of individual objects, and performance tests have verified this claim to be true.

7. Current Status

As of January 2011, the core database prototype is ready and passes the initial round of rudimentary unit tests. The API design has been completed, and the CLI interface is currently in the works. We expect to have an instance of Deska in production by the end of 2011.

8. Conclusion

We have described the general structure of the Deska database, along with an explanation about our design choices. Our approach is believed to be future-proof and enable further extension to accommodate new, emerging paradigms which come into usage at today's data centers. We recognize that there is a delicate border between designing an overly generic system which is hard to use by the real users and proposing an application which is too much tailored towards today's needs that it risks obsolescence in just a few years, essentially blocking a sustainable development.

Some of the Deska's features, like the powerful CLI interface along with support for version control and integration of various helper features, are, to our best knowledge, unique in its class of tools. We are looking forward to having a usable prototype later this year, along with the first deployment in production at the Institute of Physics in Prague.

Acknowledgments

This work was supported by the Institute of Physics of the Academy of Sciences of the Czech Republic.

References

[1] Institute of Physics of the AS CR `http://www.fzu.cz/`
[2] Kundrát J, Kerpl L, Krejčová M and Hubík T 2010 Deska: Tool for central administration of a grid site
[3] Cfengine configuration management `http://www.cfengine.org/`
[4] Mrtg: The multi router traffic grapher `http://oss.oetiker.ch/mrtg/`
[5] Nagios monitoring `http://www.nagios.org/`
[6] Ganglia monitoring system `http://ganglia.sourceforge.net/`
[7] Munin, a networked resource monitoring tool `http://munin-monitoring.org/`
[8] Ocs inventory `http://www.ocsinventory-ng.org/en/`
[9] The smurf project `https://forge.in2p3.fr/projects/smurf`
[10] Rackmonkey, a web-based tool for managing racks of equipment `http://www.rackmonkey.org/`
[11] The Deska Project Website `http://projects.flaska.net/projects/show/deska`